Assignment 3: Linearizability Testing

ECE 454: Distributed Computing

Instructor: Dr. Wojciech Golab wgolab@uwaterloo.ca

Overview

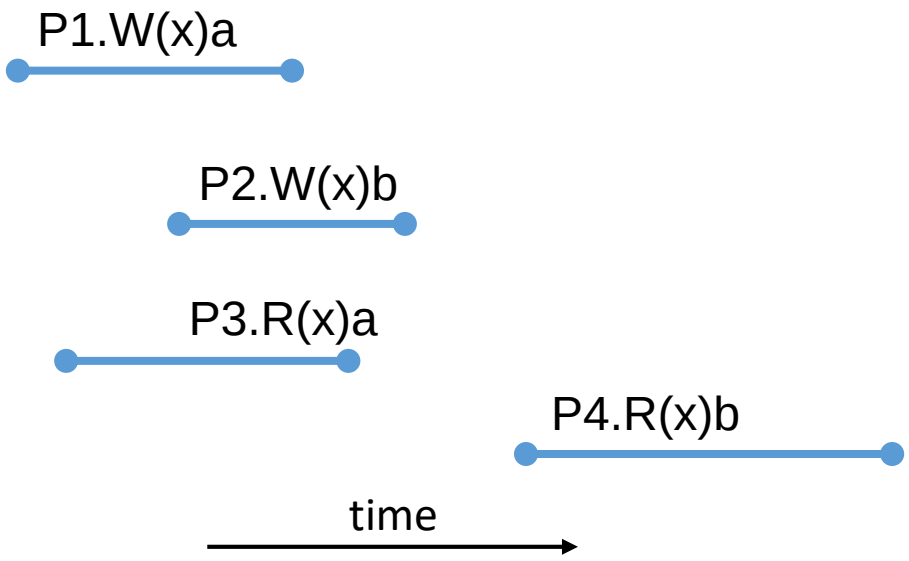
A large chunk of the grade for Assignment 3 is awarded for "correctness of outputs", which in this context is interpreted as linearizability.

This document provides instructions for linearizability testing using the techniques discussed in tutorial.

Dealing with crash failures

Our discussion of linearizability in earlier lectures assumed that an execution history is a collection of completed operations, meaning that every operation that is invoked eventually produces a response.

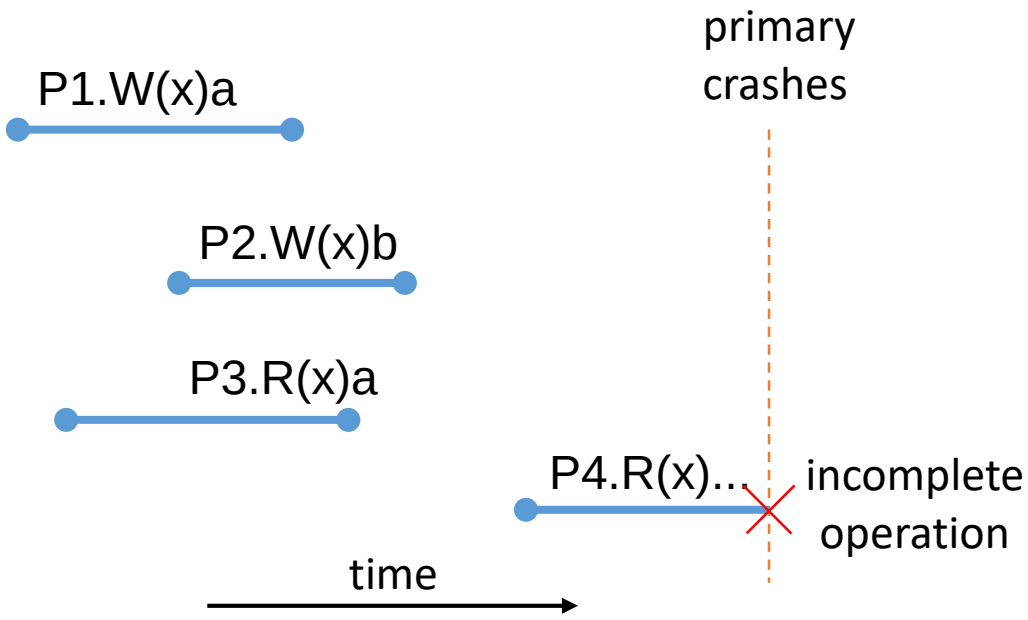
Example:



Dealing with crash failures

However, in A3 it is possible that a client invokes an operation and receives no response because the primary (or the network) fails. Such an operation is called *incomplete*.

Example:



Dealing with crash failures

Therefore, the simplified definition of linearizability given in lecture must be generalized to account for incomplete operations. We will adopt the following generalization for A3:

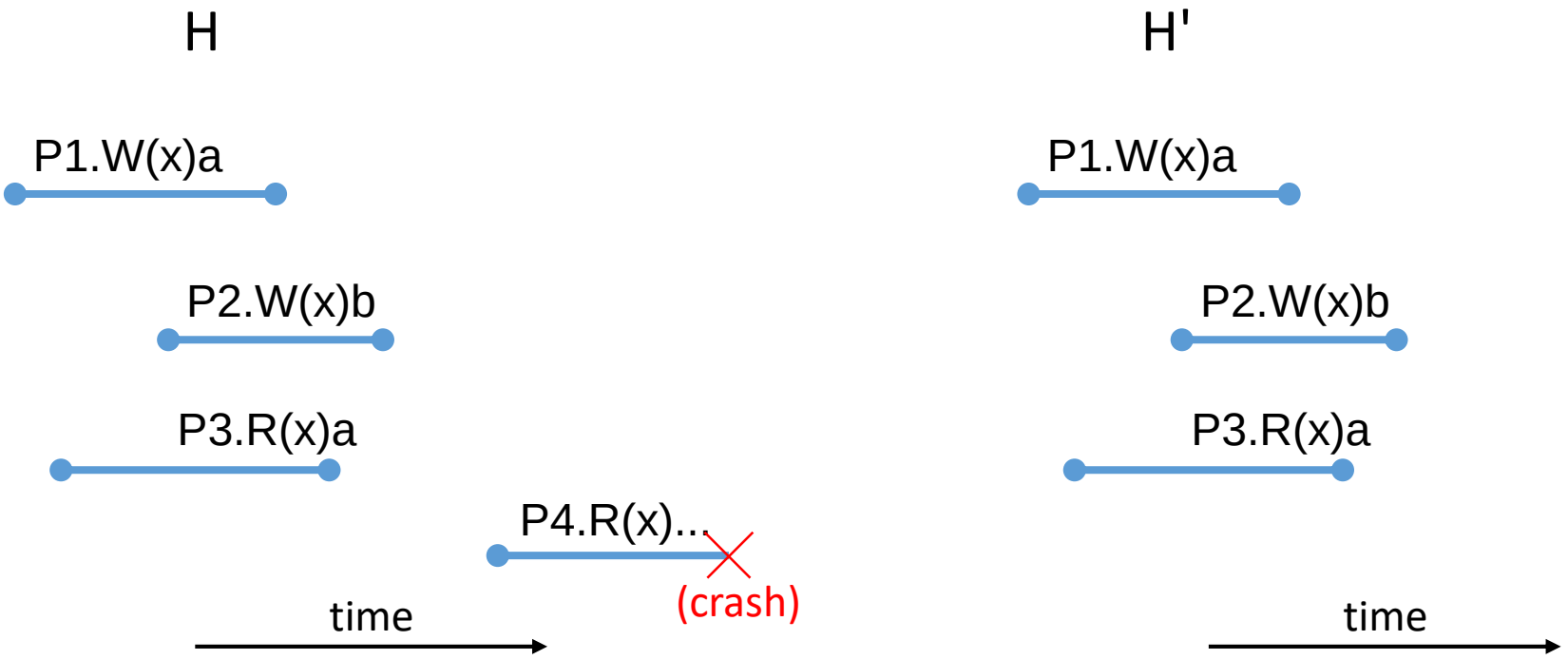
Let H be a history of read and write (i.e., get and put) operations over one or more objects (i.e., keys). Let H' be the same as H except that:

1. Every incomplete read is removed.
2. For every incomplete write, a matching response is appended at the end of the history. (Responses for different writes can be appended in any order.)

Then H is linearizable if and only if H' is linearizable.

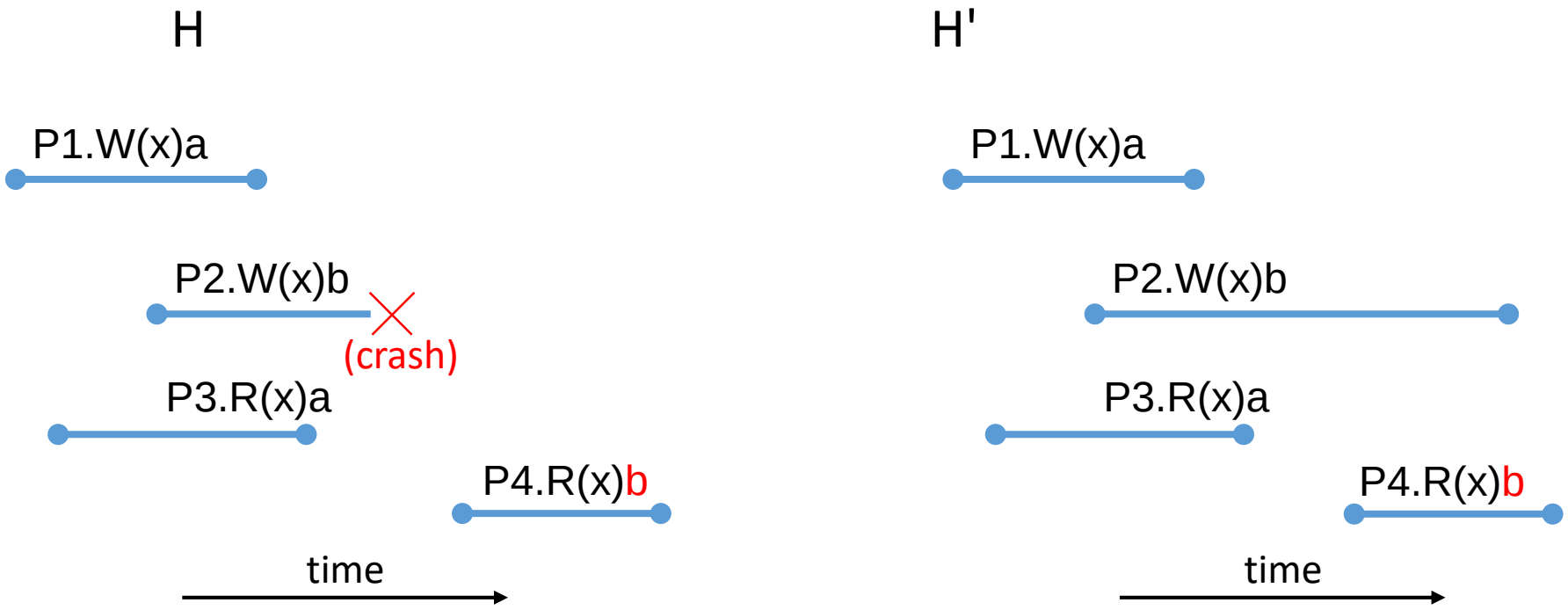
Dealing with crash failures

Example: H on the left is linearizable because H' on the right is linearizable. The interrupted read by P4 is ignored.



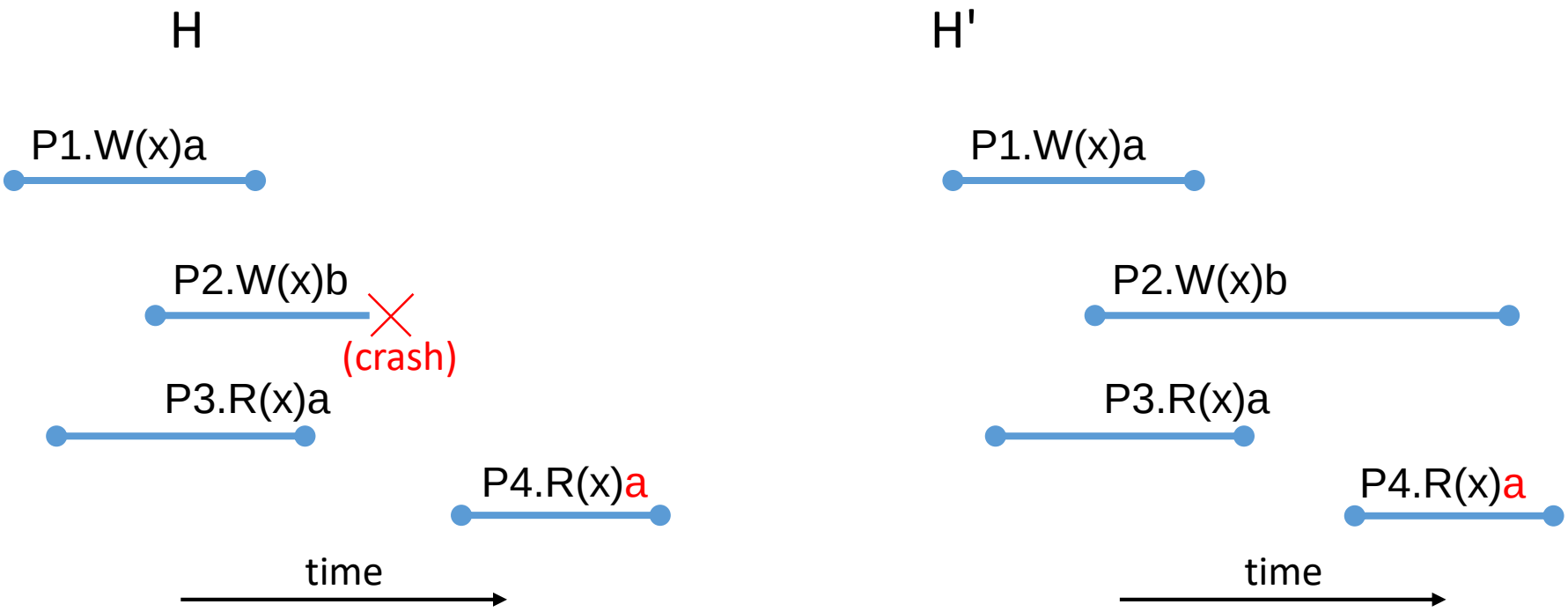
Dealing with crash failures (cont.)

Example: H on the left is linearizable because H' on the right is linearizable. The value b written by P2 is read by P4.



Dealing with crash failures (cont.)

Example: H on the left is linearizable because H' on the right is linearizable. The value b written by P2 is not read.



Instrumenting the client code

- The starter code released with A3 provides utility classes that record the history of operations executed by a client, and then analyze the history to decide linearizability.
- To use the linearizability checker, the client code is instrumented to record the invocation and response event of each storage operation. Each event is tagged with the ID of the thread that executed it.
- The linearizability checker automatically transforms the recorded history using the procedure defined earlier to deal with incomplete operations.

Instrumenting the client code

```
import ca.uwaterloo.watca.ExecutionLogger;
…
ExecutionLogger exlog = new ExecutionLogger("execution.log");
exlog.start();  // open the execution log file
…
exlog.logWriteInvocation(tid, key, value);  // record invocation
client.put(key, value);  // Thrift RPC on primary
exlog.logWriteResponse(tid); // record response
…
exlog.logReadInvocation(tid, key); // record invocation
String resp = client.get(key);  // Thrift RPC on primary
exlog.logReadResponse(tid, resp); // record response
…
exlog.stop();  // close the execution log file
```

Instrumenting the client code

- The invocation and response events are logged to a file whose name is passed to the constructor of the ExecutionLogger class. The file is human-readable and contains one line per event.
- The start() and stop() methods of the ExecutionLogger object are called exactly once, in that order, to open and close the file. The ExecutionLogger object is otherwise thread-safe, and must be shared by client threads.
- When the A3Client terminates, the runclient.sh script runs the LinearizabilityTest utility automatically on the recorded history file, and reports the number of linearizability violations.

Interpreting the output

The output of the LinearizabilityTest utility is a list of correctness scores for each key-value pair:

Key = key-40, Value = value-15, Score = 0

Key = key-40, Value = value-16, Score = 0

The scores should all be zero for a correct solution.

A score of 1 indicates a linearizability violation.

A score of 2 indicates a get operation that returned a value that is different from "" (the empty string) and was never assigned by a put operation.

Continue debugging your code until all your scores are zero!

Interpreting the output

Q: Why do I keep getting Score = 2?

A: Remember to purge the data in the key-value service before running the client, as otherwise the linearizability checker does not know what state you're starting in. For example, you can shut down the primary and backup simultaneously to purge the data.